

Detecting Incompleteness, Conflicting and Unreachability XACML Policies using Answer Set Programming

Caroline Dewi Puspa Kencana Ramli

Department of Informatics and Mathematical Modelling
 Danmarks Tekniske Universitet
 Lyngby, Denmark
 cdpu@dtu.dk

Abstract Recently, XACML is a popular access control policy language that is used widely in many applications. Policies in XACML are built based on many components over distributed resources. Due to the expressiveness of XACML, it is not trivial for policy administrators to understand the overall effect and consequences of XACML policies they have written. In this paper we show a mechanism and a tool how to analyses big access control policies sets such as (i) incompleteness policies, (ii) conflicting policies, and (iii) unreachable policies. To detect these problems we present a method using Answer Set Programming (ASP) in the context of XACML 3.0.

Keywords: XACML, access control policies, policy language, Answer Set Programming

1 Introduction

XACML (eXtensible Access Control Markup Language) is an OASIS¹ standard that describes both a policy language and a query/response language for access control policies. It has been used in many different applications range over health care information systems, transport systems to banking information systems². The policy language is used to express access control requirements (*who can access what when*) over distributed resources and the query/response language is used to query whether a particular access should be allowed (*request*) and to answer the query (*response*). Access control policies in XACML are built based on many components and combined using a particular combining algorithm.

Due to the expressiveness of XACML, it is not trivial for policy administrators to understand the overall effect and consequences of XACML policies they have written. The problem becomes more prevalent if there are no mechanisms/automated

¹ OASIS (Organization for the Advancement of Structured Information Standard) is a non-for-profit, global consortium that drives the development, convergence, and adoption of e-business standards. Information about OASIS can be found at <http://http://www.oasis-open.org/>.

² XACML references and products can be seen in <https://www.oasis-open.org/committees/download.php/42588/xacmlRefs-V1-85.html>.

tools to analyse big chunk of policies. Several problems might occur in developing access control policies such as *incomplete policies* and *conflicting policies*. Moreover, detecting *unreachable policies* might help policy administrators to remove unused policies in order to make the set of policies slimmer and make it easier to be maintained.

Analysing Incomplete Policies. It is high probable that policy developers do not define all possible situations that might occur. Incomplete access control policies might lead to a security problem. Following we present a probable scenario how an intruder can use this security hole to get an access.

In XACML, PDP (Policy Decision Point) computes a decision based on administrated policies in a database, but the final decision is made by PEP (Policy Enforcement Point). There are two PEP-biased:

1. Permit-biased PEP: if the decision from PDP is deny, then the PEP shall deny assess. All other decisions shall result in the permission of access.
2. Deny-biased PEP: if the decision from PDP is permit, then the PEP shall permit the access. All other decisions shall result in the denial of access.

In this case, there is a possibility that an intruder can get an access unintentionally by trying to query so that the response is no policy is applicable. Using Permit-biased PEP, the decision will let the intruder have access to the system.

Analysing Conflicting Policies. Conflicting policies can have serious consequences and may lead to unauthorized access. Basically, in XACML, conflicting decision never occurs since all policies are combined with a particular combining algorithm that only returns one decision. However it is interesting to analyse conflict in between policies for example different department can have different decision. By analysing conflict, the policy makers can rethink again whether they made correct policies.

Analysing Unreachable Policies. Analysing unreachable policies helps policy administrators to reduce the size of the set of policies. A policy is unreachable if for all request it never gives decision i.e., either it always not applicable or there is another policy that overrides its decision. It is safe to remove unreachable policies because their decisions never influence the final decisions.

To address the above concern we propose a logic-based XACML analysis framework using logic programs (LPs) and answer set semantics. Answer Set Programming (ASP) has become a popular approach to solve combinatorial problems declaratively. There are several efficient implementations of answer set solvers, such as ASSAT³, clasp⁴, Cmodels⁵, Smodels⁶, and many more. We present in this paper a method using ASP to solve those problems explained previously in the context of XACML 3.0 [5], the most recent version of XACML.

³ <http://assat.cs.ust.hk/>,

⁴ <http://www.cs.uni-potsdam.de/clasp/>

⁵ <http://www.cs.utexas.edu/users/tag/cmodels/>

⁶ <http://www.tcs.hut.fi/Software/smodels/>

Outline. In this paper first we explain the model and semantics of XACML 3.0 in Sect. 2. Then we describe the mapping of XACML 3.0 components into logic programs \mathcal{P}_{XACML} in Sect. 3. Next we show how to analyse access control policies such as incompleteness, conflicting and reachability XACML policies in Sect. 4. We end the paper with conclusion and future work.

2 XACML Model and Semantics

In this section we briefly describe the XACML policy language and XACML query language model. First we show the faithfully abstract syntax XACML 3.0. Then we present a semantics of XACML 3.0 without considering indeterminate values. Our argument is that we evaluate access control properties to a set of policies in statically. Hence, indeterminate values which only occur when there are errors during evaluation process do not give impact to our analysis. At the end of this section we show the semantics of XACML combining algorithms which are used for composing several access control policies.

2.1 Abstract Syntax of XACML 3.0

We summarize the syntax of XACML 3.0 in Table 1. To make the notation clear we use bold font for non-terminal **symbols**, typewriter font for terminal symbols and *identifiers* and *values* are written in italic font. Moreover, `<XACML Component>` denotes the symbol for XACML component. We use the star symbol (*) to indicate that there is zero or more of the preceding element and we use the plus symbol (+) to indicate that there is one or more of the preceding element. We assume that each policy must have a unique identifier (ID).

Table 1. Abstraction of XACML 3.0 Components

XACML Policy Components	
<code><PolicySet></code>	$:- \text{PolicySetID} = [\langle \text{Target} \rangle, \ll \text{PolicySetID}^* \gg, \text{CombID}]$ $\quad \text{PolicySetID} = [\langle \text{Target} \rangle, \ll \text{PolicyID}^* \gg, \text{CombID}]$
<code><Policy></code>	$:- \text{PolicyID} = [\langle \text{Target} \rangle, \ll \text{PolicySetID}^+ \gg \text{CombID}]$
<code><Rule></code>	$:- \text{RuleID} = [\text{Effect}, \langle \text{Target} \rangle, \langle \text{Condition} \rangle]$
<code><Condition></code>	$:- \text{propositional formulae}$
<code><Target></code>	$:- \text{Null}$ $\quad \bigwedge \langle \text{AnyOf} \rangle^+$
<code><AnyOf></code>	$:- \bigvee \langle \text{AllOf} \rangle^+$
<code><AllOf></code>	$:- \bigwedge \langle \text{Match} \rangle^+$
<code><Match></code>	$:- \text{AttrType}(\text{attribute value})$
CombID	$:- \text{po} \mid \text{do} \mid \text{fa} \mid \text{oa}$
Effect	$:- \text{deny} \mid \text{permit}$
AttrType	$:- \text{subject} \mid \text{action} \mid \text{resource} \mid \text{environment}$
XACML Request Component	
<code><Request></code>	$:- \{ \text{Attribute}^+ \}$
Attribute	$:- \text{AttrType}(\text{attribute value}) \mid \text{external state}$

There are three levels of policies in XACML, namely `<PolicySet>`, `<Policy>` and `<Rule>`. `<PolicySet>` or `<Policy>` can act as the root of a set of access control policies while `<Rule>` is a single entity that describes one particular access control policy. Through this paper, we assume that `<PolicySet>` is the root of the set of access control policies.

`<PolicySet>` and `<Policy>` have the same characteristic, i.e., they are containers for a sequence of `<PolicySet>`, `<Policy>` or `<Rule>`. A `<PolicySet>` contains either a sequence of `<PolicySet>` or a sequence of `<Policy>` while a `<Policy>` only can contains a sequence of `<Rule>`. The sequence of `<PolicySet>`, `<Policy>` or `<Rule>` is combined with a particular *combining algorithm*. There are four common combining algorithms defined in XACML 3.0, namely *permit-overrides* (po), *deny-overrides* (do), *first-applicable* (fa) and *only-one-applicable* (ooa).

A `<Rule>` describes an individual access control policy. It regulates whether an access should be *permitted* or *denied*. All `<PolicySet>`, `<Policy>` and `<Rule>` are applicable whenever their `<Target>` matches with the `<Request>`. When the `<Rule>`'s `<Target>` matches with the `<Request>`, then the applicability of the `<Rule>` is refined by its `<Condition>`.

A `<Target>` is a combination of `<Match>` elements. Each `<Match>` element describes an *attribute* that a `<Request>` should match in order to activate a policy. There are four attribute categories in XACML 3.0, namely *subject* attribute, *action* attribute, *resource* attribute and *environment* attribute. The subject attribute is the entity requesting access, e.g., a file system, a workstation, etc. The action attribute defines the type of access requested, e.g., to read, to write, to delete, etc. The resource attribute is a data, service or system components. The environment attribute can optionally provide additional information.

A `<Request>` contains a set of attributes information about access request. A `<Request>` can contain additional information such as external state condition (e.g. the current time, current temperature, etc).

2.2 XACML 3.0 Formal Semantics

The evaluation of XACML policies against a given request starts from the evaluation of `<Match>` elements and continued bottom-up until the evaluation of `<PolicySet>` as the root element. We use the $\llbracket \cdot \rrbracket$ notation to map XACML elements into their values (see the summary in Tabel 2).

Table 2. XACML Components' Values

XACML Components	Values
$\llbracket \langle \text{Match} \rangle \rrbracket$, $\llbracket \langle \text{AllOf} \rangle \rrbracket$, $\llbracket \langle \text{AnyOf} \rangle \rrbracket$, $\llbracket \langle \text{Target} \rangle \rrbracket$	match (m) and not match (nm)
$\llbracket \langle \text{Condition} \rangle \rrbracket$	true (t) and false (f)
$\llbracket \langle \text{Rule} \rangle \rrbracket$, $\llbracket \langle \text{Policy} \rangle \rrbracket$, $\llbracket \langle \text{PolicySet} \rangle \rrbracket$	permit (p), deny (d) and not applicable (na)

Evaluation of <Match> into $\{m, nm\}$. Given a <Request> Q , the evaluation of <Match> M is as follows

$$\llbracket M \rrbracket(Q) = \begin{cases} m & \text{if } M \in Q \\ nm & \text{if } M \notin Q \end{cases} \quad (1)$$

Evaluation of <AllOf> into $\{m, nm\}$. Given a <Request> Q , the evaluation of <AllOf> $A = \bigwedge_{i=1}^n M_i$ is as follows

$$\llbracket A \rrbracket(Q) = \begin{cases} m & \text{if } \forall i : \llbracket M_i \rrbracket = m \\ nm & \text{if } \exists i : \llbracket M_i \rrbracket = nm \end{cases} \quad (2)$$

where each M_i is a <Match> element.

Evaluation of <AnyOf> into $\{m, nm\}$. Given a <Request> Q , the evaluation of <AnyOf> $E = \bigvee_{i=1}^n A_i$ is as follows

$$\llbracket E \rrbracket(Q) = \begin{cases} m & \text{if } \exists i : \llbracket A_i \rrbracket = m \\ nm & \text{if } \forall i : \llbracket A_i \rrbracket = nm \end{cases} \quad (3)$$

where each A_i is a <AllOf> element.

Evaluation of <Target> into $\{m, nm\}$. Given a <Request> Q , the evaluation of <Target> $T = \bigwedge_{i=1}^n E_i$ is as follows

$$\llbracket T \rrbracket(Q) = \begin{cases} m & \text{if } \forall i : \llbracket E_i \rrbracket = m \text{ or } T = \text{Null} \\ nm & \text{if } \exists i : \llbracket E_i \rrbracket = nm \end{cases} \quad (4)$$

where each E_i is a <AnyOf> element. Empty <Target>, indicated by Null always evaluated to m.

Evaluation of <Condition> into $\{t, f\}$. Given a <Request> Q , the evaluation of <Condition> C is as follows

$$\llbracket C \rrbracket(Q) = \text{eval}(C, Q) \quad (5)$$

Note: The eval is an unspecified function that returns $\{t, f\}$.

Evaluation of <Rule> into $\{d, p, na\}$. Given a <Request> Q , the evaluation of <Rule> $R = [e, T, C]$ as follows

$$\llbracket R \rrbracket(Q) = \begin{cases} e & \text{if } \llbracket T \rrbracket(Q) = m \text{ and } \llbracket C \rrbracket(Q) = t \\ na & \text{if } (\llbracket T \rrbracket(Q) = m \text{ and } \llbracket C \rrbracket(Q) = f) \text{ or } \llbracket T \rrbracket(Q) = nm \end{cases} \quad (6)$$

where $e \in \{p, d\}$, T is a <Target> element and C is a <Condition> element.

Evaluation of <Policy> into $\{d, p, na\}$. Given a <Request> Q , the evaluation of <Policy> $P = [T, \ll R_1, \dots, R_n \gg, \text{CombID}]$ is as follows

$$\llbracket P \rrbracket(Q) = \begin{cases} na & \text{if } \llbracket T \rrbracket(Q) = nm \text{ or } \forall i : \llbracket R_i \rrbracket(Q) = na \\ \bigoplus_{\text{CombID}}(\mathbf{R}) & \text{otherwise} \end{cases} \quad (7)$$

where T is a $\langle \text{Target} \rangle$ element, and each R_i is a $\langle \text{Rule} \rangle$ element. We use \mathbf{R} to denote $\ll \llbracket R_1 \rrbracket(Q), \dots, \llbracket R_n \rrbracket(Q) \gg$.

Note: The combining algorithm denoted by \oplus_{CombID} will be explained in Sect. 2.3.

Evaluation of $\langle \text{PolicySet} \rangle$ into $\{d, p, na\}$. Given a $\langle \text{Request} \rangle Q$, the evaluation of $\langle \text{PolicySet} \rangle PS = [T, \ll P_1, \dots, P_n \gg, \text{CombID}]$ is as follows

$$\llbracket PS \rrbracket(Q) = \begin{cases} na & \text{if } \llbracket T \rrbracket(Q) = nm \text{ or } \forall i : \llbracket P_i \rrbracket(Q) = na \\ \oplus_{\text{CombID}}(\mathbf{P}) & \text{otherwise} \end{cases} \quad (8)$$

where T is a $\langle \text{Target} \rangle$ element and each P_i is a $\langle \text{Policy} \rangle$ (or $\langle \text{PolicySet} \rangle$) element. We use \mathbf{P} to denote $\ll \llbracket P_1 \rrbracket(Q), \dots, \llbracket P_n \rrbracket(Q) \gg$.

2.3 XACML Combining Algorithms

There are four common combining algorithms defined in XACML 3.0, namely permit-overrides (po), deny-overrides (do), first-applicable (fa) and only-one-applicable (ooa). The permit-overrides combining algorithm takes permit decision as the most priority than deny decision while the deny-overrides combining algorithm takes deny decision over permit. Likewise their names, the first-applicable combining algorithm return the first $\langle \text{Rule} \rangle$ (or $\langle \text{Policy} \rangle$ or $\langle \text{PolicySet} \rangle$) that is applicable (either permit (p) or deny(d) value) and the only-one-applicable combining algorithm return a decision whenever only one $\langle \text{Rule} \rangle$ (or $\langle \text{Policy} \rangle$ or $\langle \text{PolicySet} \rangle$) which is applicable, otherwise it returns not applicable (na).

Permit-Overrides The permit-overrides combining algorithm is intended for those cases where a permit decision should have priority over a deny decision.

Let $S = \ll v_1, v_2, \dots, v_n \gg$ be a sequence of policy values. The permit-overrides combining algorithm, \oplus_{po} , is defined as follows

$$\oplus_{\text{po}}(S) = \begin{cases} p & \text{if } \exists i : v_i = p \\ d & \text{if } \forall i : v_i \neq p \text{ and } \exists j : v_j = d \\ na & \text{otherwise} \end{cases} \quad (9)$$

Deny-Overrides The deny-overrides is the mirror of permit-overrides whereas the deny decision has more priority over a permit decision.

Let $S = \ll v_1, v_2, \dots, v_n \gg$ be a sequence of policy values. The deny-overrides combining algorithm, \oplus_{do} , is defined as follows

$$\oplus_{\text{do}}(S) = \begin{cases} d & \text{if } \exists i : v_i = d \\ p & \text{if } \forall i : v_i \neq d \text{ and } \exists j : v_j = p \\ na & \text{otherwise} \end{cases} \quad (10)$$

Let $S = \ll v_1, v_2, \dots, v_n \gg$ be a sequence of policy values. The first-applicable combining algorithm, \oplus_{fa} , is defined as follows

(11)

Let $S = \ll v_1, v_2, \dots, v_n \gg$ be a sequence of policy values. The only-one-applicable combining algorithm, \oplus_{ooa} , is defined as follows

(12)

First we explain the syntax of logic program (LP) in this section. Then we show the transforming XACML 3.0 into LP. The semantics of LP is explained in the following section when we use it for analysis purposes.

We start by introducing some notations and terminologies which we will use through the paper. **First-Order Language.** We consider an *alphabet* consisting of (finite or countably infinite) disjoint sets of variables, constants, function symbols, predicate symbols, connectives $\{\mathbf{not}, \wedge, \leftarrow\}$, punctuation symbols $\{“(”, “”, “”)\}$ and special symbols $\{\top, \perp\}$. In this paper we will use upper case letters to denote variables and lower case letters to denote constants, function and predicate symbols. Terms, atoms, literals and formulae are defined as usual. The *language* given by an alphabet consists of the set of all formulae constructed from the symbols occurring in the alphabet.

(13)

where A is either an atom or \perp and each B_i , $1 \leq i \leq n$, is an atom or \top . \top is a valid formula. A is called the *head* and $B_1 \wedge \dots \wedge B_m \wedge \mathbf{not} B_{m+1} \wedge \dots \wedge \mathbf{not} B_n$ is called

body of the clause. We usually write $B_1 \wedge \dots \wedge B_m \wedge \text{not } B_{m+1} \wedge \dots \wedge \text{not } B_n$ simply as $B_1, \dots, B_m, \text{not } B_{m+1}, \dots, \text{not } B_n$.

We refer the rule as a *constraint* when A is \perp . One should observe that the body of a rule must not be empty. A rule of the form $A \leftarrow \top$ is called a *fact*.

A *logic program* (LP) is a finite set of rules. $\text{ground}(\mathcal{P})$ denotes the set of all ground instances of the program \mathcal{P} .

3.2 XACML Transformations

The transformation of XACML components is based on the semantics of each component explained in Sect. 2.2. First we recall the syntax of each component then we show how the transformation is.

<Request> Transformation. *XACML Syntax:* Let $Q = \{A_1, \dots, A_n\}$, $1 \leq i \leq n$, be a <Request> component. The transformation of <Request>, Q , into LP \mathcal{P}_Q is as follows

$$A_i \leftarrow \top. \quad 1 \leq i \leq n$$

<Match> Transformation. *XACML Syntax:* Let M be a <Match> component. The transformation of <Match> M into LP \mathcal{P}_M is as follows (see (1) for <Match> evaluation)

$$\begin{aligned} \text{val}(M, m) &\leftarrow M. \\ \text{val}(M, nm) &\leftarrow \text{not } M. \end{aligned}$$

<AllOf> Transformation. *XACML Syntax:* Let $A = \bigwedge_{i=1}^n M_i$ be an <AllOf> component where each M_i is a <Match> component. The transformation of <AllOf> A into LP \mathcal{P}_A is as follows (see (2) for <AllOf> evaluation)

$$\begin{aligned} \text{val}(A, m) &\leftarrow \text{val}(M_1, m), \dots, \text{val}(M_n, m). \\ \text{val}(A, nm) &\leftarrow \text{val}(M_i, nm). \quad (1 \leq i \leq n) \end{aligned}$$

<AnyOf> Transformation. *XACML Syntax:* Let $E = \bigvee_{i=1}^n A_i$ be an <AnyOf> component where each A_i is an <AllOf> component. The transformation of <AnyOf> E into LP \mathcal{P}_E is as follows (see (3) for <AnyOf> evaluation)

$$\begin{aligned} \text{val}(E, m) &\leftarrow \text{val}(A_i, m). \quad (1 \leq i \leq n) \\ \text{val}(E, nm) &\leftarrow \text{val}(A_1, nm), \dots, \text{val}(A_n, nm). \end{aligned}$$

<Target> Transformation. *XACML Syntax:* Let $T = \bigwedge_{i=1}^n E_i$ be a <Target> component where each E_i is an <AnyOf> component. The transformation of <Target> T into LP \mathcal{P}_T is as follows (see (4) for <Target> evaluation)

$$\begin{aligned} \text{val}(T, m) &\leftarrow \text{val}(E_1, m), \dots, \text{val}(E_n, m). \\ \text{val}(\text{null}, m) &\leftarrow \top. \\ \text{val}(T, nm) &\leftarrow \text{val}(E_i, nm). \quad (1 \leq i \leq n) \end{aligned}$$

<Condition> Transformation. *XACML Syntax:* We assume that the <Condition> element is a boolean formula which the evaluation of <Condition> is based on eval function. The transformation of <Condition> C into LP \mathcal{P}_C is as follows

$$\text{val}(C, V) \leftarrow \text{eval}(C, V).$$

In our previous example of $\langle \text{Rule} \rangle r1$, the $\langle \text{Condition} \rangle \text{cond}(r1)$ is $\text{patient.id}(X) \wedge \text{patient_record.id}(X)$. The possibility of eval function is like following

```

 $\mathcal{P}_{\text{cond}(r1)} :$ 
 $\text{val}(\text{cond}(r1), V) \leftarrow \text{eval}(\text{cond}(r1), V).$ 
 $\text{eval}(\text{cond}(r1), t) \leftarrow \text{patient\_id}(X), \text{patient\_record\_id}(X).$ 
 $\text{eval}(\text{cond}(r1), f) \leftarrow \text{patient\_id}(X), \text{patient\_record\_id}(Y), X \neq Y.$ 

```

The $\text{error}(\text{patient_id}(X))$ and $\text{error}(\text{patient_record_id}(X))$ indicate possible errors might occur, e.g., the system could not connect to the database so that the system does not know the ID of the patient.

$\langle \text{Rule} \rangle$ Transformation. *XACML Syntax:* Let $R_{id} = [E, T, C]$ be a $\langle \text{Rule} \rangle$ component where $E \in \{p, d\}$, T is a $\langle \text{Target} \rangle$ and C is a $\langle \text{Condition} \rangle$. The transformation of $\langle \text{Rule} \rangle R_{id}$ into LP $\mathcal{P}_{R_{id}}$ is as follows (see (6) for $\langle \text{Rule} \rangle$ evaluation)

```

 $\text{val}(R_{id}, E) \leftarrow \text{val}(T, m), \text{val}(C, t).$ 
 $\text{val}(R_{id}, na) \leftarrow \text{val}(T, m), \text{val}(C, f).$ 
 $\text{val}(R_{id}, na) \leftarrow \text{val}(T, nm).$ 

```

$\langle \text{Policy} \rangle$ Transformation. *XACML Syntax:* Let $P_{id} = [T, \ll R_1, \dots, R_n \gg, \text{CombID}]$ be a $\langle \text{Policy} \rangle$ component where T is a $\langle \text{Target} \rangle$, $\langle R_1, \dots, R_n \rangle$ be a sequence of $\langle \text{Rule} \rangle$ elements and CombID be a combining algorithm identifier. In order to indicate that the $\langle \text{Policy} \rangle$ contains $\langle \text{Rule} \rangle R_i$, thus for every $\langle \text{Rule} \rangle R_i$ contained in $P_{id} = [T, \ll R_1, \dots, R_n \gg, \text{CombID}]$, $\mathcal{P}_{P_{id}}$ also contains:

$$\text{dec}(P_{id}, R_i, E) \leftarrow \text{val}(R_i, E). (1 \leq i \leq n)$$

Next, we do a transformation for $\langle \text{Policy} \rangle P_{id}$ and add into LP $\mathcal{P}_{P_{id}}$ is as follows (see (7) for $\langle \text{Policy} \rangle$ evaluation)

```

 $\text{val}(P_{id}, na) \leftarrow \text{val}(T, nm).$ 
 $\text{val}(P_{id}, na) \leftarrow \text{val}(R_1, na), \dots, \text{val}(R_n, na).$ 
 $\text{val}(P_{id}, E) \leftarrow \text{val}(T, m), \text{dec}(P_{id}, R, V), V \neq na, \text{algo}(\text{CombID}, P_{id}, E).$ 

```

We write formulae $\text{dec}(P_{id}, R, V), V \neq na$ to make sure that there is a $\langle \text{Rule} \rangle$ in the $\langle \text{Policy} \rangle$ that is not evaluated to na. We do this to avoid a return value from a combining algorithm that is not na even though all of the $\langle \text{Rule} \rangle$ elements are evaluated to na.

$\langle \text{PolicySet} \rangle$ Transformation. The transformation of $\langle \text{PolicySet} \rangle$ is similar to the transformation of $\langle \text{Policy} \rangle$ component.

XACML Syntax: Let $PS_{id} = [T, \ll P_1, \dots, P_n \gg, \text{CombID}]$ be a $\langle \text{Policy} \rangle$ component where T is a $\langle \text{Target} \rangle$, $\langle P_1, \dots, P_n \rangle$ be a sequence of $\langle \text{Policy} \rangle$ (or $\langle \text{PolicySet} \rangle$) elements and CombID be a combining algorithm identifier. The transformation of $\langle \text{PolicySet} \rangle PS_{id}$ into logic program $\mathcal{P}_{PS_{id}}$ is as follows For every $\langle \text{Policy} \rangle$ (or $\langle \text{PolicySet} \rangle$) contained in $PS_{id} = [T, \ll P_1, \dots, P_n \gg, \text{CombID}]$, $\mathcal{P}_{PS_{id}}$ also contains:

$$\text{dec}(PS_{id}, P_i, E) \leftarrow \text{val}(P_i, E). (1 \leq i \leq n)$$

And we following rules into $\mathcal{P}_{PS_{id}}$

$$\begin{aligned} \text{val}(PS_{id}, na) &\leftarrow \text{val}(T, nm). \\ \text{val}(PS_{id}, na) &\leftarrow \text{val}(P_1, na), \dots, \text{val}(P_n, na). \\ \text{val}(PS_{id}, E) &\leftarrow \text{val}(T, m), \text{dec}(PS_{id}, P, V), V \neq na, \text{algo}(\text{CombID}, PS_{id}, E). \end{aligned}$$

3.3 Combining Algorithm Transformation

We use P for an variable of $\langle \text{Policy} \rangle$ identifier and R, R_1 and R_2 for variables of $\langle \text{Rule} \rangle$ identifiers. In case the evaluation of $\langle \text{PolicySet} \rangle$, the input P is for $\langle \text{PolicySet} \rangle$ identifier, R, R_1 and R_2 are for $\langle \text{Policy} \rangle$ (or $\langle \text{PolicySet} \rangle$) identifiers.

Permit-Overrides Transformation. Let \mathcal{P}_{po} be a LP obtained by permit-overrides combining algorithm transformation (see (9) for the permit-overrides combining algorithm semantics). \mathcal{P}_{po} contains:

$$\begin{aligned} \text{algo}(po, P, p) &\leftarrow \text{dec}(P, R, p). \\ \text{algo}(po, P, d) &\leftarrow \text{not algo}(po, P, p), \text{dec}(P, R, d). \\ \text{algo}(po, P, na) &\leftarrow \text{not algo}(po, P, p), \text{not algo}(po, P, d). \end{aligned}$$

Deny-Overrides Transformation. Let \mathcal{P} be a LP obtained by deny-overrides combining algorithm transformation (see (10) for the permit-overrides combining algorithm semantics). \mathcal{P}_{po} contains:

$$\begin{aligned} \text{algo}(po, P, d) &\leftarrow \text{dec}(P, R, d). \\ \text{algo}(po, P, p) &\leftarrow \text{not algo}(po, P, d), \text{dec}(P, R, p). \\ \text{algo}(po, P, na) &\leftarrow \text{not algo}(po, P, d), \text{not algo}(po, P, d). \end{aligned}$$

First-Applicable Transformation. Let \mathcal{P}_{fa} be a logic program obtained by first-applicable combining algorithm transformation (see (11) for the first-applicable combining algorithm semantics). For each $\langle \text{Policy} \rangle$ (or $\langle \text{PolicySet} \rangle$) which uses first-applicable combining algorithm, $P_{id} = [T, \ll R_1, \dots, R_n \gg, fa]$, $\mathcal{P}_{P_{id}}$ contains:

$$\begin{aligned} \text{algo}(fa, P, E) &\leftarrow \text{dec}(P, R_1, E), E \neq na. \\ \text{algo}(fa, P, E) &\leftarrow \text{dec}(P, R_1, na), \text{dec}(P, R_2, E), E \neq na. \\ &\vdots \\ \text{algo}(fa, P, E) &\leftarrow \text{dec}(P, R_1, na), \dots, \text{dec}(P, R_{n-1}, na), \\ &\quad \text{dec}(P, R_n, E). \end{aligned}$$

Only-One-Applicable Transformation. Let \mathcal{P}_{ooa} be a logic program obtained by only-one-applicable combining algorithm transformation (see (12) for the only-one-applicable combining algorithm semantics). \mathcal{P}_{ooa} contains:

$$\begin{aligned} \text{not_one_applicable}(P) &\leftarrow \text{dec}(P, R1, X), \text{dec}(P, R2, Y), R1 \neq R2, X \neq na, Y \neq na. \\ \text{algo}(ooa, P, E) &\leftarrow \text{dec}(P, R, E), \text{not not_one_applicable}(P). \\ \text{algo}(ooa, P, na) &\leftarrow \text{not_one_applicable}(P). \end{aligned}$$

4 Policy Analysis

We use the semantics of LP \mathcal{P}_{XACML} – the result from transforming XACML components into series of LPs – to analyse access control policy properties. In this section, we present three policy analysis cases namely analysing on incompleteness policies, conflicting policies and unreachable policies. The completeness and free of conflict properties have been introduced by Samarati and di Vimercati in [7] and formalized using Belnap four-valued logic [2] by Bruns and Huth in [3]. In this section we show how we present ASP programs to capture those properties⁷. Our intention is to have an automatic tool that shows XACML formalization and in the same time it can be used to help policy administrators to analyse their policies sets.

4.1 Semantics of Logic Programs

The declarative semantics of a logic program is given by a model-theoretic semantics of formulae in the underlying language. The formal definition of answer set semantics can be found in many literatures like in [1,4].

Interpretations and Models The *Herbrand Universe* $\mathcal{U}_{\mathcal{L}}$ for a language \mathcal{L} is the set of all ground terms that can be formed from the constants and function symbols appearing in \mathcal{L} . The *Herbrand base* $\mathcal{B}_{\mathcal{L}}$ for a language \mathcal{L} is the set of all ground atoms that can be formed by using predicate symbols from \mathcal{L} and ground terms from $\mathcal{U}_{\mathcal{L}}$ as arguments. By $\mathcal{B}_{\mathcal{P}}$ we denote the Herbrand base for language underlying the program \mathcal{P} . When the context is clear, we are safe to omit \mathcal{P} .

An *interpretation* I of a program \mathcal{P} is a mapping from the Herbrand base $\mathcal{B}_{\mathcal{P}}$ to the set of truth value true and false ($\{\top, \perp\}$). All atoms belong to interpretation I are mapped to \top . All atoms which does not occur in I are mapped to \perp .

The truth value of arbitrary formulae under some interpretation can be determined from a truth table as usual (see Table 3).

Table 3. Truth Values for Formulae

ϕ	ψ	$\text{not } \phi$	$\phi \wedge \psi$	$\phi \leftarrow \psi$
\top	\top	\perp	\top	\top
\top	\perp	\perp	\perp	\top
\perp	\top	\top	\perp	\perp
\perp	\perp	\top	\perp	\top

The logical value of ground formulae can be derived from Table 3 in the usual way. A formula ϕ is then *true under interpretation* I , denoted by $I(\phi) = \top$, if all its ground instances are true in I ; it is *false under interpretation* I , denoted by $I(\phi) = \perp$, if there is a ground instance of ϕ that is false in I .

Let I be an interpretation. I *satisfies* formula ϕ , denoted by $I \models \phi$, if $I(\phi) = \top$. For a program \mathcal{P} , we say I *satisfies* of \mathcal{P} , denoted by $I \models \mathcal{P}$, if I satisfies for every clause in \mathcal{P} .

⁷ We call ASP programs for logic programs with answer set semantics.

Let \mathcal{I} be a collection of interpretations. Then an interpretation I in \mathcal{I} is called *minimal* in \mathcal{I} if and only if there is no interpretation J in \mathcal{I} such that $J \subsetneq I$. An interpretation I is called *least* in \mathcal{I} if and only if $I \subseteq J$ for any interpretation J in \mathcal{I} . A model M of a program \mathcal{P} is called *minimal* (respectively *least*) if it is minimal (respectively least) among all models of \mathcal{P} .

The answer set semantics of logic program \mathcal{P} assigns to \mathcal{P} a collection of *answer sets* – interpretations of $\text{ground}(\mathcal{P})$. An interpretation I of $\text{ground}(\mathcal{P})$ is an answer set for \mathcal{P} if I is minimal (w.r.t. set inclusion) among the interpretations satisfying the rules of

$$\mathcal{P}^I = \{A \leftarrow B_1, \dots, B_m \mid A \leftarrow B_1, \dots, B_m, \text{not } B_{m+1}, \dots, \text{not } B_n \in \mathcal{P} \text{ and } I(\text{not } B_{m+1}, \dots, \text{not } B_n) = \text{true}\}$$

A logic program can have a unique, many or none answer set(s). Therefore, we show that programs with a particular characteristic are guaranteed to have unique answer set.

Acyclic Programs. We say that a program is *acyclic* when there is no cycle in the program. The acyclicity in the program is guaranteed by the existence of a certain fixed assignment of natural numbers to atoms that is called a *level mapping*.

A *level mapping* for a program \mathcal{P} is a function

$$l : \mathcal{B}_{\mathcal{P}} \rightarrow \mathbf{N}$$

where \mathbf{N} is the set of natural numbers and $\mathcal{B}_{\mathcal{P}}$ is the Herbrand base for \mathcal{P} . We extend the definition of level mapping to a mapping from ground literals to natural numbers by setting $l(\text{not } A) = l(A)$.

Let \mathcal{P} be a logic program and l be a level mapping for \mathcal{P} . \mathcal{P} is *acyclic with respect to l* if for every clause $A \leftarrow B_1, \dots, B_m, \text{not } B_{m+1}, \dots, \text{not } B_n$ in $\text{ground}(\mathcal{P})$ we find

$$l(A) > l(B_i) \text{ for all } i \text{ with } 1 \leq i \leq n$$

\mathcal{P} is *acyclic* if it is acyclic with respect to some level mapping.

Acyclic programs are guaranteed to have unique answer sets [1].

4.2 XACML Semantics Based On ASP Semantics

We can see from Sect. 3 that all of the XACML 3.0 transformation programs are acyclic. Thus, it is guaranteed that $\mathcal{P}_{\text{XACML}}$ has unique answer set.

Proposition 1. Let $\mathcal{P}_{\text{XACML}}$ be a program obtained from XACML 3.0 element transformations and let \mathcal{P}_Q be a program transformation of $\langle \text{Request} \rangle Q$. Let I be the answer set of $\mathcal{P}_{\text{XACML}} \cup \mathcal{P}_Q$. Then the following equation holds

$$\llbracket X \rrbracket(Q) = V \text{ iff } \text{val}(X, V) \in I$$

4.3 Analysis on Incompleteness Policies

A set of policies is *complete* if it always returns a decision given for any request. XACML defines that there is one `<PolicySet>` as the root of a set of policies. Therefore, we formally express *completeness property* as follows:

$$\mathbf{complete}: \forall Q : \llbracket PS_{root} \rrbracket(Q) \neq \text{na}$$

where Q for `<Request>` and PS_{root} is the root of `<PolicySet>` element in the set of policies.

We say that there is a *gap* in the policy set if it is not complete. Hence, we formally express *gap property* as follows:

$$\mathbf{gap}: \neg \mathbf{complete}$$

It is equal to

$$\mathbf{gap}: \exists Q : \llbracket PS_{root} \rrbracket(Q) = \text{na}$$

The idea of having gap property is to have a logic program that can show answer sets whenever there is gap in the policies. We use the answer sets as the witnesses of the incompleteness policies.

In order to check gap property we should generate all possible values restored in the database for each attribute. Each attribute only possible to have one value. Thus, we use *cardinality constraint* [8,10] and the encoding is as follows:

$$\begin{aligned} \mathcal{P}_{generate_one} : \\ 1\{subject(X) : subject_db(X)\}1 & \leftarrow \top. \\ 1\{action(X) : action_db(X)\}1 & \leftarrow \top. \\ 1\{resource(X) : resource_db(X)\}1 & \leftarrow \top. \\ 1\{environment(X) : environment_db(X)\}1 & \leftarrow \top. \end{aligned}$$

The intuitive meaning of the above cardinality constraints is that, for each subject in the database, exactly one instance of subject request is generated. The conversion holds for other attributes.

We say there is a gap whenever we can find a request that makes value of the PS_{root} is na. Here is the encoding:

$$\begin{aligned} \mathcal{P}_{gap} : \\ gap & \leftarrow \text{val}(PS_{root}, \text{na}). \\ \perp & \leftarrow \mathbf{not} \ gap. \end{aligned}$$

We force ASP solver to find a gap by putting a constraint $\perp \leftarrow \mathbf{not} \ gap$.

The answer sets of program $\mathcal{P} = \mathcal{P}_{XACML} \cup \mathcal{P}_{generate_one} \cup \mathcal{P}_{gap}$ are the witnesses that the set of policies encoded in \mathcal{P}_{XACML} is incomplete. When there is no model satisfies the program then we are sure that the set of policies captures all of possible cases.

4.4 Analysis on Conflicting Policies

A conflict never occurs in XACML because the structure of policies where there is only one `<PolicySet>` as the root of all of policies and all of others policies are combined by combining algorithm. Each combining algorithm returns a single decision either permit or deny and never return both decisions in the same time. However, it is still interesting to know whether there are two `<Rule>` give conflict decisions. We formally define a conflict is as follows:

$$\text{conflict: } \exists Q : \llbracket R \rrbracket(Q) = p \wedge \llbracket R' \rrbracket(Q) = d$$

In order to compute whether there is a conflict in the set of policies, we encode a logic program for conflict property as follows:

$$\begin{aligned} \mathcal{P}_{\text{conflict}} : \\ \text{conflict} &\leftarrow \text{val}(R, p), \text{val}(R', d), R \neq R'. \\ \perp &\leftarrow \text{not conflict}. \end{aligned}$$

The same as gap condition, we force ASP solver to find a conflict by putting a constraint $\perp \leftarrow \text{not conflict}$.

A conflict can be analysis whenever $\mathcal{P} = \mathcal{P}_{\text{XACML}} \cup \mathcal{P}_{\text{generate_one}} \cup \mathcal{P}_{\text{conflict}}$ returns answer sets. The returning models are evidences where the conflict between `<Rule>` occurs. We conclude that a set of policies is conflict-free if and only if program \mathcal{P} is unsatisfied, i.e., there is no returned model.

4.5 Analysis on Reachability Policies

A policy is reachable if there is a request such that the decision is made based on this policy. Usually in a big set of policies, there is a policy that is not reachable. This happens because policies are built based on several components and combined together. We formally define a reachability property as follows:

$$\text{reachable}(R): \exists Q : \llbracket R \rrbracket(Q) \neq \text{na}.$$

where Q is `<Request>` element and R is `<Rule>` element.

The encoding of reachability property in logic program is as follows:

$$\begin{aligned} \mathcal{P}_{\text{reachable}} : \\ \text{reachable}(R) &\leftarrow \text{val}(R, E), E \neq \text{na}. \end{aligned}$$

Formally, a policy is not reachable if for every request either:

1. It always return na.

$$\text{unreachable}(R): \forall Q : \llbracket R \rrbracket = \text{na}$$

2. in case of permit-overrides combining algorithm, a policy is not reachable if its decision is deny but the final decision of the root policy is permit.

$$\text{unreachable: } \forall Q : \llbracket R \rrbracket = d \wedge \llbracket P \rrbracket(Q) = p$$

where in $P = [T, \ll \dots, R, \dots \gg, po]$

3. In case of deny-overrides combining algorithm, a policy is not reachable if its decision is permit but the final decision of the root policy is deny.

$$\text{unreachable}(\mathbf{R}): \forall Q : \llbracket R \rrbracket = p \wedge \llbracket P \rrbracket(Q) = d$$

where in $P = [T, \ll \dots, R, \dots \gg, \text{do}]$

4. In case of only-one-applicable combining algorithm, a policy is not reachable if it is applicable policy but the final decision of the root policy is not applicable. This indicates that there is another policy that is also applicable.

$$\text{unreachable}(\mathbf{R}): \forall Q : \llbracket R \rrbracket \neq \text{na} \wedge \llbracket P \rrbracket(Q) = \text{na}$$

where in $P = [T, \ll \dots, R, \dots \gg, \text{ooa}]$

5. In case of first-applicable combining algorithm, a policy is not reachable if it is applicable but there is another policy in the same collection that is in the earlier of the sequence that is also applicable.

$$\text{unreachable}(R_j): \forall Q : \llbracket R_j \rrbracket \neq \text{na} \wedge \llbracket R_i \rrbracket(Q) \neq \text{na} \wedge i < j$$

where in $P = [T, \ll \dots, R_i, \dots, R_j, \dots \gg, \text{fa}]$

First of all we should generate all possible attributes. This time, the encoding is different with program $\mathcal{P}_{\text{generate_one}}$ because we want to generate all possible attributes, not only one. Hence, we do not use cardinality constraint in this encoding. Here is the encoding:

```

 $\mathcal{P}_{\text{generate\_all}} :$ 
 $\text{subject}(X) \leftarrow \text{subject\_db}(X).$ 
 $\text{action}(X) \leftarrow \text{action\_db}(X).$ 
 $\text{resource}(X) \leftarrow \text{resource\_db}(X).$ 
 $\text{environment}(X) \leftarrow \text{environment\_db}(X).$ 

```

Following we translate each unreachable condition into logic program

```

 $\mathcal{P}_{\text{not\_reachable}} :$ 
 $\text{not\_reachable}(R) \leftarrow \text{not\_reachable}(R).$ 
 $\text{not\_reachable}(R) \leftarrow \text{val}(P, p), \text{dec}(P, R, d).$ 
 $\text{not\_reachable}(R) \leftarrow \text{val}(P, d), \text{dec}(P, R, p).$ 
 $\text{not\_reachable}(R) \leftarrow \text{val}(P, \text{na}), \text{dec}(P, R, E), E \neq \text{na}.$ 

```

In the case of first-applicable combining algorithm, there is a possibility a policy returns permit and the final decision is also permit, but, the permit of the final decision comes from the earlier applicable policy. Hence, we should take care of the ordering of policies. We need to add extra rules in the program transformations such as: for every $\langle \text{Rule} \rangle R_i$ contained in $P_{id} = [T, \ll R_1, \dots, R_n \gg, \text{fa}]$, $\mathcal{P}_{P_{id}}$ also contains:

$$\text{dec}(P_{id}, R_i, E, I) \leftarrow \text{val}(R_i, E). (1 \leq i \leq n)$$

Here we add to our $\mathcal{P}_{not_reachable}$

$$not_reachable(R_j) \leftarrow dec(P_{id}, R_i, E, I), dec(P_{id}, R_j, E', J), E \neq na, E' \neq na, I < J.$$

To check unreachable property we add to our program $\mathcal{P}_{not_reachable}$:

$$\begin{aligned} not_reachable &\leftarrow not_reachable(R). \\ \perp &\leftarrow \mathbf{not} \ not_reachable. \end{aligned}$$

We force ASP solver to find unreachable policies by putting a constraint $\perp \leftarrow \mathbf{not} \ not_reachable$. When $not_reachable(R)$ is in the answer set of $\mathcal{P} = \mathcal{P}_{XACML} \cup \mathcal{P}_{generate_all} \cup \mathcal{P}_{reachable} \cup \mathcal{P}_{not_reachable}$ then it is safe to remove policy R from the set because it is unreachable.

5 Conclusion and Future Work

We have shown a mechanism to map XACML 3.0 components into logic programs. Using the advantages of ASP technique to solve combinatorial problems efficiently we have presented ASP programs to capture analysing in access control policies incompleteness property, conflicting property and unreachability property. Our intention is to have an automatic tool that both showing formalization and also help policy administrators to analyse their policies sets.

For future work, we would like to analyse conflict in attribute based like in Singh's work [9]. We also would like to extend our work to handle Role-Based Access Control (RBAC) [6] and see the conflict might occurs between different roles.

In order to reduce the policies, we could inspect redundancy between policies. We should find a subset of policies that might capture the whole possible decisions might happen in all policies. Thus, we could have smaller set than the original policies set.

References

1. Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, February 2003.
2. N.D. Belnap. A useful four-valued logic. In G. Epstein and J.M. Dunn, editors, *Modern Uses of Multiple-Valued Logic*, pages 8–37. D. Reidel, Dordrecht, 1977.
3. Glenn Bruns and Michael Huth. Access-control via belnap logic: Effective and efficient composition and analysis. In *21st IEEE Computer Security Foundations Symposium*, June 2008.
4. Michael Gelfond. Handbook of knowledge representation. In B. Porter F. van Harmelen, V. Lifschitz, editor, *Foundations of Artificial Intelligence*, volume 3, chapter Answer Sets, pages 285–316. Elsevier, 2007.
5. Erik Rissanen. eXtensible Access Control Markup Language (XACML) version 3.0 (committe specification 01). Technical report, OASIS, <http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-cd-03-en.pdf>, August 2010.

6. Erik Rissanen. Xacml v3.0 core and hierarchical role based access control (rbac) profile version 1.0 (committe specification 01). Technical report, OASIS, <http://docs.oasis-open.org/xacml/3.0/xacml-3.0-rbac-v1-spec-cs-01-en.pdf>, August 2010.
7. Pierangela Samarati and Sabrina de Capitani di Vimercati. Access control: Policies, models, and mechanisms. In *Foundations of Security Analysis and Design, Tutorial Lectures*, volume 2171 of *Lecture Notes in Computer Science*, pages 137–196. Springer Verlag, 2001.
8. Patrik Simons, Ilkka Niemelá, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.
9. Kamalbir Singh and Sarbjeet Singh. Design and evaluation of XACML conflict policies detection mechanism. *International Journal of Computer Science and Information Technology*, 2:65–74, 2010.
10. Tommi Syrjänen. *Lparse 1.0 User's Manual*.